VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace konečných prvků
Visualization of finite elements

2010 Bc. Roman Truhlář

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 23. dubna 2010 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2010 .....

Děkuji tímto vedoucímu diplomové práce Ing. Petr Gajdoš Ph.D., za odbornou pomoc a konzultaci při realizaci tohoto projektu.

Abstrakt

Tato práce se zabývá vizualizací výsledků metody konečných prvků a výběrem a implementací vhodné prostorové datové struktury pro rychlé výpočty řezů. Obsahuje také některé další optimalizační techniky pro efektivnější správu paměti a cachování dat na disk. To vše je zakomponováno do existující aplikace. Nechybí ani finální porovnání rychlosti zvolené datové struktury s variantou bez ní.

Klíčová slova: vizualizace, konečné prvky, simulace, fyzikální vlastnosti, octree, správa paměti, cache, OpenGL, rozsáhlá data, datová struktura, elementy, geometrie, grafika

Abstract

This Diploma Thesis describes visualization of finite elements and choice and implementation of suitable space-partitioning data structure for fast object clipping. Includes several other optimalization techniques for more effective memory management and caching data to disc. All of that is merged into existing application. Final comparison of performance with and without chosen data structure is presented at the end.

Keywords: visualization, finite elements, simulation, material properties, octree, memory management, cache, OpenGL, large data sets, data structure, elements, geometry, graphics

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
bsp tree	–	binární strom pro rekurzivní rozdělování prostoru
kd-tree	–	k-dimenzionální strom
octree	–	oktantový strom
R-tree	–	datová struktura pro indexaci prostoru
shader	–	počítačový program určený pro zpracování přímo na grafické kartě

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

1.1 Motivace pro zobrazování konečných prvků

Metoda konečných prvků je numerická metoda sloužící k simulaci průběhů napětí, deformací, vlastních frekvencí, proudění tepla, jevů elektromagnetismu, proudění tekutin a tak dále na vytvořeném fyzikálním modelu. Její princip spočívá v diskretizaci spojitého prostoru do určitého (konečného) počtu prvků, přičemž zjišťované parametry jsou určovány v jednotlivých uzlových bodech. Pro ukázku, jak vypadá taková vizualizace konečných prvků, je Obrázek 1, ve kterém jsou zobrazeny tlaky v jednotlivých místech konstrukce modelu ojnice.

1.2 Cíl práce

Cílem práce je navrhnout a implementovat vhodnou datovou strukturu a metody pro efektivní vizualizaci konečných prvků. Důraz je kladen na základní geometrické operace nad rozsáhlými daty.
Základní body zadání:

1. Přehled existujících postupů a řešení při vizualizaci konečných prvků.
2. Návrh a implementace zvoleného řešení.
3. Integrace řešení do komplexního vizualizačního software.
4. Výkonnostní testy.

Obrázek 1: Ojnice s vizualizovanými tlaky

2 Vizualizace konečných prvků

K vizualizaci této metody jsou zapotřebí nejen vstupní data, ale i výběr vizualizační metody. Má práce byla soustředěna k rasterizaci, ale aplikace měla umožňovat i vizualizovat konečné prvky volumetrickou metodou, která je obsahem diplomové práce mého kolegy Bc. Lubomíra Duraje.

2.1 Vstupní geometrická data vizualizace

Vstupní data lze získat ze dvou různých zdrojů. Tím primárním je přes připojení k webové službě poskytující tyto data ke stahnutí a nebo ze souboru, pokud je uložený na disku. Nejdříve se získají data reprezentující vrcholy modelu. Ty jsou uloženy za sebou v poli jako tři double hodnoty pro každý vrchol. Následně se zpracují data s elementy, které jsou uloženy v poli a každý element začíná s identifikátorem domény, do které patří (v této práci dále nazívaný jako meshpart), typem elementu (viz. Obrázek 4), počtem vrcholů, a seznamem indexů do pole vrcholů, které definují unikátní vrcholy, ze kterých se element skládá. Geometrie každého elementu, respektive pro vybudování indexů pro jeho geometrie, má na pevno nadefinováno pořadí v poli indexů elementu pro konkrétní typy elementů. S těmito informacemi již lze model vykreslit.

Dříve zmiňovaným meshpartem se rozumí nějaká ucelená část modelu, například v modelu s kuličkovým ložiskem by byla každá koule samostatný meshpart a stejně tak i obě kruhové konstrukce.

2.2 Výstupy metody konečných prvků

Pro vizualizaci konečných prvků se navíc používá další hodnota pro každý vrchol. Ta nese nějakou fyzikální informaci v různých podobách od jednoho skaláru až po n-dimenzionální tensory [**?**]. Každý model může mít více těchto dat a proto jsou odděleny od dat modelu a jsou samostatně ke stáhnutí z databáze. Aplikace umožnuje pracovat se dvěma typy výsledků metody konečných prvků. Tím prvním jsou výsledky, které různě deformují objekt a tím druhým jsou výsledky mapovatelné na barvu. Současně může být na jednom modelu aktivovaný pouze jeden výsledek pro každý tento typ. Při mapování na barvu se nejdříve zjistí tzv. bounding box ohraňující všechny vstupní výsledky a přepočítá se na složku texturové souřadnice podle Equation 1. Jednotlivé texturové souřadnice takto odpovídají hodnotám box_min pro 0 a box_max pro 1 pro každou složku souřadnice.

$$texture\_coord = (actual\_result - box\_min)/(box\_max - box\_min) \tag{1}$$

Uvedený vzorec je obecně platný, ale pro výsledky metody konečných prvků o například pěti skalárech produkuje pěti-rozměrnou texturovou souřadnici a nic takového nelze v praxi použít. Proto se i tyto vícerozměrné texturové souřadnice dále mapují na jedno-rozměrný interval $< 0, 1 >$.

Výsledná texturová souřadnice pak slouží k určení výsledné barvy v daném bodě modelu z gradientní textury, kterou lze libovolně upravovat.

2.3 Vysvětlení a řešení problému

Jelikož vstupní vizualizovaný model je již tvořen trojúhelníky, není jeho samotná vizualizace takový problém. To neznamená ale, že stačí pouze nastavit ukazatel na body a říct vybranému grafickému API "vykresli n trojúhelníků". Nejenom by mohlo dojít k problému, kdy nedostačuje velikost systémové paměti, o které se bude dále věnovat sekce 3.2, ale i k přílišnému zatížení grafické karty počítače, zaviněné renderováním obrovského množství geometrií. Tento problém částečně řeší preprocessing na geometriích popisovaný v sekci 3.1.2.

2.4 Metody vizualizace

Vizualizací se rozumí způsob, jak zobrazit data na výstupní zařízení jako je monitor. K tomuto účelu existují následující dvě základní metody.

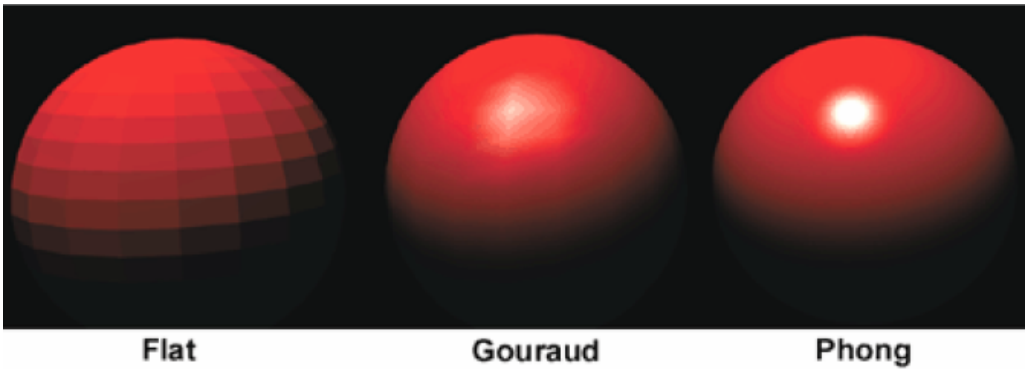
2.4.1 Rasterizace

Rasterizace je velmi často používanou technikou pro svou jednoduchost a hlavně použitelnost. Je hardwarově implementována na všech grafických kartách a proto je velmi rychlá. Základní myšlenka spočívá v geometrické interpretaci jakéhokoliv objektu použitím trojúhelníků (ikdyž prakticky lze použít i úsečky a čtyřúhelníky). Každý, takto složený objekt, se nachází v tzv. Object space. V tomto prostoru je objekt, když se vytvoří v nějakém grafické programu, či se získá z různých snímacích zařízení. V zásadě se jedná o souřadnicový systém, ve kterém je objekt na svém počátku při načtení do programu. V tomto prostoru by ale všechny objekty byly na sobě, většinou někde na souřadnicích [0,0,0], a proto následuje další tzv. ModelView Space [1], který kombinuje umístění objektu ve scéně a pak posun a otočení scény do pohledu kamery. Objekt je ale stále ve 3D prostoru, zatímco monitor dokáže zobrazit pouze 2D objekty a k tomuto účelu slouží tzv. Projection space, ve kterém se promítne 3D scéna na 2D plátno (XY plocha). Ve většině případů se používá perspektivní promítání, ve kterém vzdálené objekty vypadají menší než ty blízké. Pro všechny objekty, které se nakonec použijí k zobrazení musí splňovat:

- $-1.0 <= x <= 1.0$
- $-1.0 <= y <= 1.0$
- $-1.0 <= z <= 0.0$

Podle použitého rozlišení se scéna roztáhne podél x a y os o polovinu rozlišení, posune a vertikálně převrátí, aby levý horní roh měl souřadnice [0,0]. Výsledné souřadnice bodů jsou pak přímé 2D souřadnice v okně a lze je již zobrazit. Jednotlivé prostory jsou realizované transformačními maticemi, kterými se každý bod každého trojúhelníku vynásobí podle Equation 2.

[1] Tato terminologie se používá v OpenGL, v DirectX je zvykem používat dva prostory namísto jednoho, World & View space



Obrázek 2: Ukázka rozdílu mezi jednotlivými stínovacími technikami

$$clip\_position = Projection * ModelView * object\_position \tag{2}$$

Tento proces se vykonává na grafické kartě ve vertex shaderu. Poté se po ploše, kterou pokrývá právě zpracovávaný trojúhelník, vypočítá barva podle zvoleného stínovacího modelu [Lan10]. Rozdíly mezi němi jsou z Obrázek 2 velmi zřejmé.

- **Konstantní stínování** - pro každý trojúhelník se vypočítá barva a jí se pokryje celý povrh. Velmi nerealistické, jelikož zvírazňuje hrany trojúhelníků, ale v průmyslovém odvětví stále použitelný.
- **Gouraudovo stínování** - barva je vypočítaná na každém vertexu a pak lineárně interpolována po povrchu trojúhelníku. Známe také jako "per-vertex lightning".
- **Phongovo stínování** - barva se vypočítává na každém zobrazovaném pixelu. Známe také jako "per-pixel lightning".

O samotném výpočtu barvy by bylo možné napsat knihu. Běžně používaná metoda počítá barvu podle Phongova osvětlovacího modelu dle Equation 3

$$I = k_a i_a + \sum_{m \in lights} s_m c_{att} (k_d (L_m * N) i_d + k_s (R_m * V)^{\alpha} i_s) \tag{3}$$

$$c_{att} = \frac{1}{att_x + att_y d + att_z d^2} \tag{4}$$

k jsou koeficienty materiálu (indexy: a = ambient, d = diffuse, s = specular), i jsou barvy právě procházeného světla. i_d a i_s jsou často spojeny do jedné barvy a i_a je jedna globální barva pro ambientní světlo. Exponent α specifikuje velikost odrazu. N je normála v daném bodě, L_m je vektor z renderovaného bodu do světla m, R_m je odražený vektor světla od bodu podle normály a nakonec V je vektor směru do kamery. Všechny tyto vektory musí být normalizované. c_att snižuje intenzitu světla se vzdáleností od světla podle Equation 4. s_m udává, zda je bod ve stínu. Při hodnotě 1 je plně osvícen, při 0 je mezi

bodem a světlem nějaká překážka. Jeho výpočet je velmi náročný a musí být výpočítaný v samostatném kroku mimo hlavní vykreslovací smyčku [2]. Implementace stínů ale v této práci není důležitá.

2.4.2 Volumetrické renderování

Jedná se o techniku, která zobrazuje 2D projekcí 3D diskrétních navzorkovaných dat v podobě mnoha paralelních 2D obrazů. Obecně existuje více metod k tomuto typu vizualizace, jako jsou například tzv. shear warp a splatting techniky [Kau05]. V této aplikaci je použitá ale technika Volume ray casting, která poskytuje obraz vysoké kvality. Hrubý popis, jak tato metoda funguje, spočívá ve vystřelování paprsků z pozice kamery přes všechny pixely výsledného obrazu (podobné jako u vizualizace přes raytracing[3]). Každý paprsek pak prochází jednotlivými 2D paralelními pláty a každý průchod plátem přispívá k výsledné barvě pixelu podle zvolené tzv. transfer function. Pro představu může jít například o funkci, která pokaždé vezme 75% původní barvy a 25% nové (první vrstva by byla vyjímkou, která by měla na počátku 100% příspěvek).

[2] Stíny se řeší technikami jako Shadow Mapping a Shadow Volume

[3] Další typ vizualizace podávající velmi reálné obrazy, obvzláště u lesklých objektů

3 Implementace

Jelikož jsem navazoval na práci svého vedoucího diplomové práce, musel jsem začít seznámením se s tehdejší verzí implementace a navázat na ní. Celá práce je rozdělená na sedm různých projektů.

1. **Database** - Komunikace se serverem, na kterém jsou uloženy vizualizované modely. Pro mou část práce nebylo potřeba zde vůbec zasahovat.
2. **DataCore** - Zde byl kladen hlavní důraz pro mou práci. Zde jsou veškeré třídy pro správu dat a operacemi nad nimi.
3. **GGradientWidget** - Okno s nastavením mapování výsledků metody konečných prvků na barvy.
4. **Manager** - Hlavní spouštěná aplikace s GUI. Pouze menší zásahy do tohoto projektu z mé strany. Hlavně úpravy pro selekci geometrií.
5. **MeshDataCore** - Obsahuje třídy obstarávající samotnou vizualizace modelů, včetně metod pro selekci geometrií.
6. **QGLViewer** - Knihovna postavená nad Qt knihovnou pro usnadnění spojení s OpenGL.
7. **Renderer** - Widget realizující renderování rasterizací.
8. **VolumeRenderer** - Widget realizující renderování volumetrickou metodou.

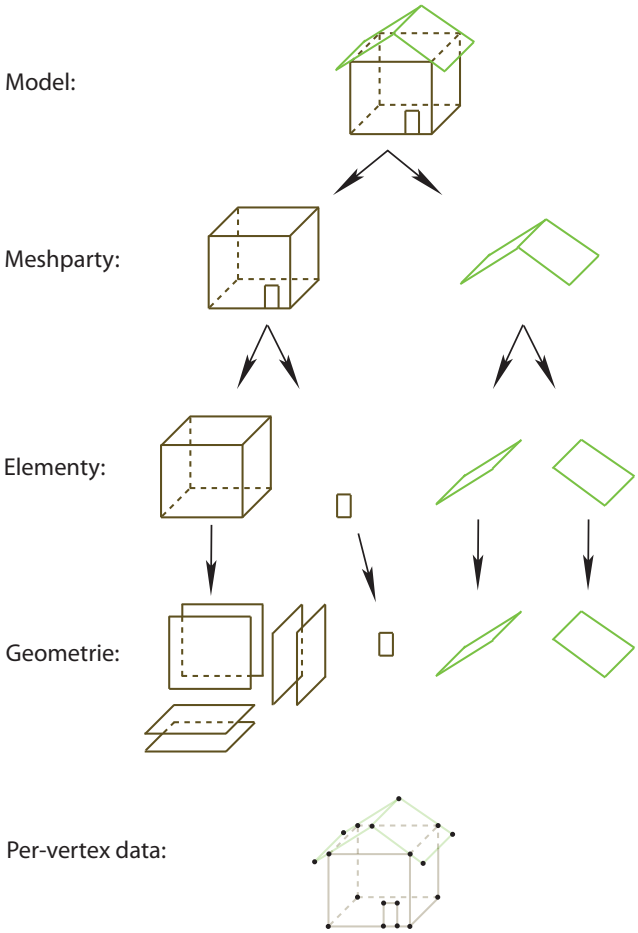
3.1 Hierarchie dat

Pro vizualizaci rasterizací jakéhokoliv objektu je zapotřebí znát nejrůznější data, která tvoří daný objekt. Jak bylo ukázáno v sekci 2.4, objekt se musí skládat z malých geometrických objektů jako jsou trojúhelníky nebo rovinné čtyřúhelníky. V této práci se objekt skládá navíc i z elementů tvořících strukturu konečných prvků. Detailní návaznost jednotlivých složek modelu zobrazuje Obrázek 3 na modelu jednoduchého domečku.

Všechny tyto informace k objektu jsou podrobněji rozepsané v následujících sekcích.

3.1.1 Per vertex data

K tomuto typu dat patří informace ke každému vertexu v modelu. Vždy se nejméně jedná o pozice vertexu, ale pouze tato samotná informace nedovoluje počítat se složitějším osvětlovacím modelem než je obecně konstantní barva pro každý trojúhelník (nepočítáli se texturování). Bohužel je toto jediná informace ze vstupních dat, kterou máme v počátku k dispozici, a proto je nutné v preprocessingu dopočítat některé další jako jsou hlavně vertex normály, díky kterým můžeme model už mnohem reálněji nasvítit. Jejich výpočet není složitý, což dokazuje i pseudokód 1.



Obrázek 3: Návaznost částí modelu

```
n = number of vertices
normals = array[n]
foreach t in triangles do:
  calculate face normal fn by cross product (v1−v0)∗(v2−v0)
  normalize fn
  normals[index of v0] += fn
  normals[index of v1] += fn
  normals[index of v3] += fn

foreach normal in normals do:
  normalize normal
```

Výpis 1: Pseudokód výpočtu vertex normál

Často bývají další per-vertex data bitangenty a binormaly umožňující pokročilejší grafické efekty jako bump mapping a pod. Pro technické účely této práce ale nejsou potřeba. Jinak je to ale s texturovými souřadnicemi, které slouží k samplování gradientní 1D textury, vygenerované pro účely mapovaní výsledků metody konečných prvků na barvu. Jak přesně tato metoda funguje, bylo popsáno v sekci 2.2.

3.1.2 Geometrie

Každý objekt se může skládat ze základních primitiv jako jsou úsečky, trojúhelníky nebo čtyřúhelníky a každé takové primitivum musí nést informace o svých bodech, v jakých elementech se nachází (o elementech více v sekci 3.1.3) a značku definující hlavně o jaký typ geometrie se jedná a další doplňkové, například zda je vyselektován, či se jedná o hraniční geometrii. Přesně k tomuto pak slouží jednoduchá datová struktura GeometryInfo podle Listing 2

```
class GeometryInfo
{
public:
  GeometryFlag flag;
  int          elements[2];
  unsigned int pIndices[4];
};
```

Výpis 2: Datová struktura GeometryInfo

Vzhledem k opravdu velkému množství geometrií, ze kterých se skládá model, je důležité tuto třídu napsat co nejefektivněji pro alokaci a současně co nejmenší pro minimalizaci potřebné paměti. Proto proměnná na indexy bodů je konstatní pole, které pojme až 4 indexy. Je zřejmé, že např. pro trojúhelník by stačilo pole o velikosti 3, takže by se zdálo, že bychom ušetřili 4 bajty paměti za každý trojúhelník v modelu. Bohužel to ale není pravda. Vyžadovalo by to dynamickou alokaci paměti podle typu geometrie, která by značně snížila rychlost alokace a hlavně následné dealokace této třídy, a také by byl potřeba navíc jeden ukazatel na toto pole, takže pro náš příklad s trojúhelníkem by byla velikost v paměti stejná (3 indexy * 4 bajty + 1 ukazatel * 4 bajty). Pouze pro úsečky by došlo na plýtvání pamětí. Ty ovšem nejsou natolik časté v modelu, aby ve výsledku znamenali nějaký problém. Další optimalizace spočívá ve speciální alokaci této třídy přes memory manager, který byl naimplementován k co nejrychlejší alokaci velkého počtu stejných objektů a o kterém je věnovaná samostatná sekce 3.3.

V modelu se může obecně nacházet obrovská množství těchto geometrií v řádu až 100k, kde větší část je ale schovaná vně objektu, a tak stejně nejsou vidět a není důvod grafické kartě o nic vůbec říkat (vyjímkou je wireframe zobrazení, kde je potřeba kreslit opravdu všechno). Proces zjišťování, zda je geometrie schovaná uvnitř nebo na okraji objektu se skládá ze dvou částí:

1. **Výpočet hodnot proměnné elements v každé geometrii.**
   Při načítání modelu má každý element své unikátní geometrie, ikdyž sousedící elementy sdílejí stejnou plošku a měla by tak být pouze jedna instance této geometrie. A právě toto opravuje pseudokód 3. Na počátku tedy má každá geometrie

pouze v elements[0] validní index. Nejdříve se vypočítá pro všechny geometrie jejich hash, aby se nemuselo kontrolovat vždy všechny indexy do pole bodů, a tak se poznalo, zda jsou dvě geometrie stejné. Poté se toto pole hashů seřadí spolu s polem geometrií, aby se dostali stejné geometrie vedle sebe v poli a současně aby platilo, že například pátý index v poli hashů odpovídal páté geometrii. V praxi to znamená, že při třídění, pokud přehodíme dva hashe v poli hashů, tak přehodíme také dvě geometrie na stejných indexech v poli. Nakonec se projdou všechny hashe a porovnává se vždy hash s tím následujícím v poli, a pokud se rovnají tak se jedná o stejné geometrie dvou různých elementů a proto se sjednotí nastavením g1.elements[1] = g2.elements[0] a přeskočí se je v další iteraci v cyklu. A protože druhá geometrie je redundantní, tak ji lze úplně vypustit ze seznamu, pokud se samozřejmě upraví index v elementu z té druhé geometrie na tu první. Setříděním geometrií se znehodnotili indexy geometrií v elementech a je proto nutné je aktualizovat.

```
hashes = array()
foreach geometries g do:
  append to hashes calculated hash from g
Sort(hashes, geometries);
for i = 0 to hash.size() − 1 do:
  update element at geometries[i].elements[0] geometry index
  if hashes[i] == hashes[i+1]:
    geometries[i].elements[1] = geometries[i+1].elements[0]
    mark geometries[i+1] as redundant
    update element at geometries[i+1].elements[0] geometry index
    ++i # to skip redundant geometry
```

Výpis 3: Pseudokód výpočtu indexů elementů pro každou geometrii

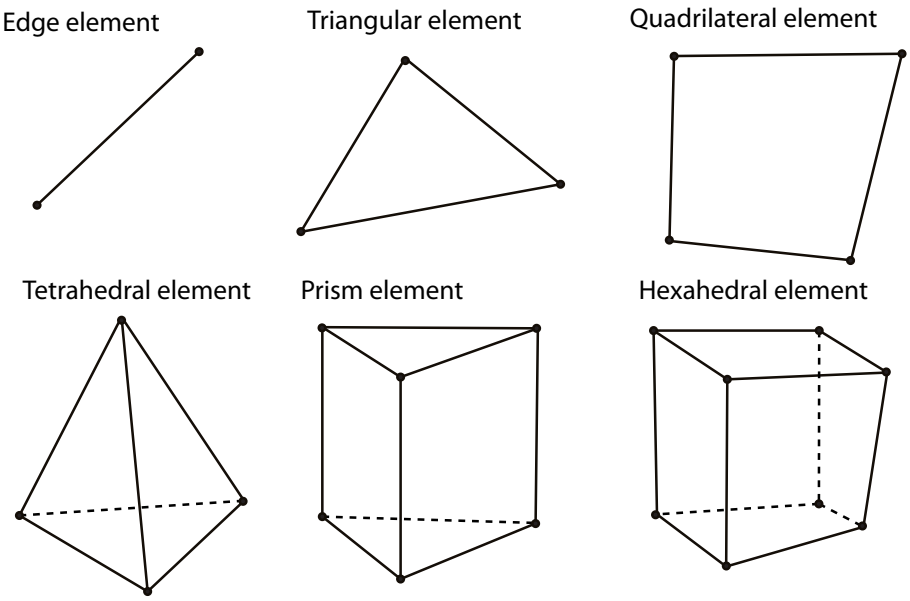
2. Nalezení okrajových geometrií.
Tento krok je již velmi snadný, jelikož spočívá pouze v průchodu všemi geometriemi a přečtení hodnoty elements[1], která v případě, že je objekt uvnitř objektu, musí být validní index do pole elementů. Geometrie uvnitř objektu je vždy vlastněna dvěmi elementy, které se dotýkají právě přes tuto geometrii. Pseudokód tohoto kroku je v následujícím textu 4.

```
foreach geometry g do:
  if g.elements[1] != −1:
    add boundary mark to g.flag
```

Výpis 4: Pseudokód nalezení okrajových geometrií

3.1.3 Elementy

Metoda konečných prvků nepočítá na úrovni samotné geometrie, ale nad jistými podobjekty nazývanými v této práci jako elementy. Každý takový element je tvořen několika



Obrázek 4: Typy elementů

geometriemi a sousední elementy vždy sdílí stejnou geometrii na svých společných plochách. K dispozici je celkem 6 typů elementů, které jsou znázorněny v Obrázek 4.

Pro ilustraci, jak vypadá tato datová struktura elementu, jsem uvedl základní podobu třídy v Listing 5. RendererElementType definuje typ elementu, podle kterého pak lze získat i množství geometrií, ze kterých se element skládá a tak i velikost následného dynamického pole, ve kterém jsou indexy do pole geometrií.

```
class ElementInfo
{
public:
  RendererElementType ret;
  unsigned int*       g;
};
```

Výpis 5: Datová struktura ElementInfo

3.1.4 Meshparty

Každý model se skládá z několika ucelených částí, které dohromady tvoří celý model. Ten je tak rozdělen do tzv. domén, kde každý definuje z jakých elementů se skládá. Protože jsou ale elementy přerozděleny a vlastněny vždy jen jedním meshpartem a sousedící elementy z různých meshpartů sdílí společnou geometrii, musí tato geometrie být duplikována. A tak meshpart nejen drží své elementy, ale i jeho geometrie a per-vertex

data. Samotný proces rozdělení dat modelu do meshpartů je popsán v Listing 6, kde je popsán hrubě bez nějakých optimalizací jen pro představu.

```
foreach mp in meshpart array do:
    vert_rewrite_map[] = {−1}
    ei_rewrite_map[] = {−1}
    # add all elements of mp to its array
    foreach ei in element info array do:
      if ei.domain == mp:
        append ei to mp
        ei_rewrite_map[ index of ei ] = index of appended ei in mp

    # add all geometries of mp to its array
    foreach gi in geometry info array do:
      if gi belongs to mp:
        # create new copy of gi. It 's changed here and original settings is
        # necessary to know
        new_gi = new geometry info(gi)
        new_gi.elements[0] = ei_rewrite_map[ gi.elements[0] ]
        append index of new_gi to element at mp.elements[ gi.elements[0] ]
        if gi.elements[1] != −1:
          new_gi.elements[1] = ei_rewrite_map[ gi.elements[1] ]
          append index of new_gi to element at mp.elements[ gi.elements[1] ]

        append new_gi to mp.gi array
        append index of mp.gi to geometryIndex[new_gi.flag] array

        # add only new points and normals to mp array
        foreach v in new_gi.indices do:
          if vert_rewrite_map[v] == −1:
            vert_rewrite_map[v] = number of points in mp
            append new point at v to mp
            append new normal at v to mp
          new_gi.indices[v] = vert_rewrite_map[v]
          append vert_rewrite_map[v] to geometry[new_gi.flag] array
    # save resuls to file
    serialize mp
```

Výpis 6: Rozdělení dat modelu do meshpartů

Po tomto procesu se smažou veškerá data, která platila pro celý model před rozdělením. Teprve nyní je každý meshpart absolutně osamostatněn od ostatních a lze jej velmi snadno a rychle serializovat a deserializovat.

3.1.5 Model

Model reprezentuje již celý, z databáze či souboru, načtený vizualizovaný objekt. Sám osobě představuje pouze datové úložiště všech meshpartů, které převzali roli správců dat modelu.

3.2 Cache systém

Dnešní počítače mají omezenou systémovou pamět, do které se nemusí vlézt celý vizualizovaný model se všemi příslušnými daty. A proto je nezbytné použít nějakou komplexnější správu těchto dat. Řešením je ukládat data na disk a v případě potřeby je načíst do paměti, tzv. proces serializace a deserializace. Nabízí se tak použití nějakého automatizovaného systému, který by pří přístupu do dat se podíval, zda jsou v paměti a pokud ne tak je tam načetl. Využít tak jakého si handlu k datům. Tímto způsobem by se navenek schovala ona serializace a usnadnil se přístup k datům. Nevýhoda tohoto přístupu je, že by vyžadovala obrovské množství požadavků na I/O operace, které by efektivně snížili odezvu a rychlost aplikace, které jsou v této práci velmi důležité. Navíc pro renderování, je potřeba mít všechna data, jako jsou body a normály, celá v paměti v jednom bloku a ve stejném pořadí, aby indexy geometrie stále ukazovali na správné body, a protože prakticky mohou indexy vyžadovat body z nejrůznějších míst onoho pole bodů, bylo by vždy potřeba mít v paměti všechny body a normály a byly bychom defakto opět na začátku. Použitelné by to bylo pouze pro jiná než per-vertex data, jako jsou elementy a geometrie.

Jelikož je ale model rozdělen na několik částí (meshparty), lze vstupní data přerozdělit a načíst vždy všechna data pro každý meshpart z disku najednou. Ve výsledku se takto přečte téměř stejné množství dat pro vykreslení celého modelu, ale pouze s n I/O operací, kde n je počet meshpartů, namísto m operací u předchozího způsobu, kde je potřeba I/O operace na každý chybějící bod v paměti a podobně, a proto platí $n < m$. Při přerozdělování dat do meshpartů je ale nezbytné některá data duplikovat, jako jsou body, které vyžadují dva a více meshpartů, nebo geometrie na společných hranicích dvou meshpartů. Nejedná se však o nějaké kritické množství dat a pokud tento malý nárůst v celkově vyžadovaném místě na disku velmi urychlý běh aplikace, tak je to přijatelná cena.

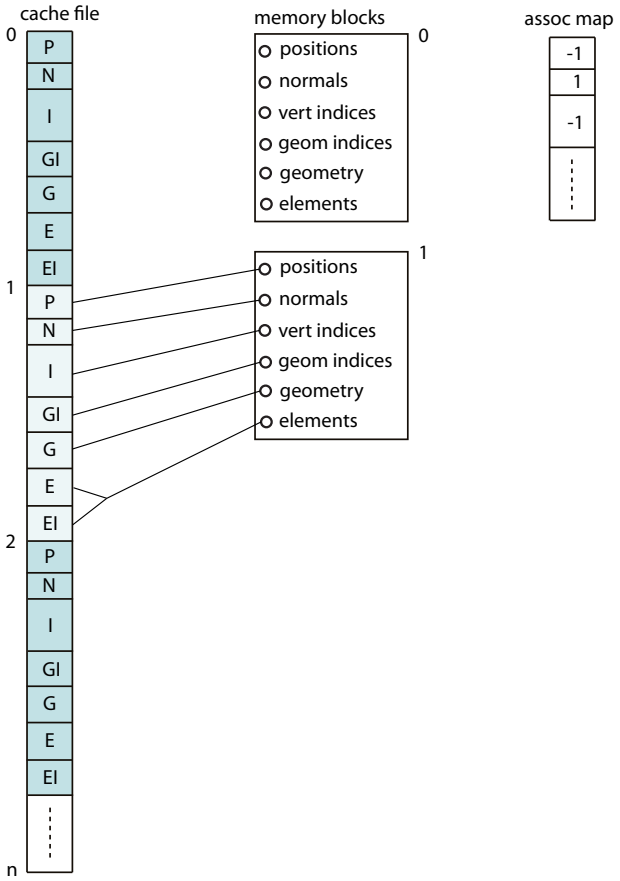
A proto pro implementaci byl zvolen tento druhý způsob, který ale vyžaduje relativně náročný preprocessing přerozdělovávání dat do meshpartů, ale odvděčí se svou rychlostí při deserializacích a celkově vyžadovaná pamět se změní na celkově vyžadované místo na disku.

3.2.1 Princip

Serializované meshparty jsou uložené zasebou v souboru na disku a cache systém si ukládá jednotlivé offsety od počátku souboru pro každý meshpart. Jelikož je k dispozici většinou více systémové paměti, než kolik potřebuje jeden libovolný meshpart, udržuje tento cache systém několik největších meshpartů permanentně v systémové paměti, aby omezil jednotlivé I/O operace pro každý deserializační dotaz. Počet těchto meshpartů se spočítá zhruba podle Equation 5. V podstatě se se z volné fyzické systémové paměti vezme 75% a vypočítá se, kolik mespartů o největší velikosti v daném modelu by se do paměti vešlo. Nejedná se o příliš přesné vyjádření tohoto počtu, ale to ani není možné z důvodu, že dopředu neznáme kolik paměti budou potřebovat další části aplikace.

$$keep\_in\_sys\_mem = (avail\_sysmem * 0.75)/meshpart\_max\_size \tag{5}$$

Jakmile je znám počet meshpartů, které mohou zůstat v paměti, je vytvořeno n instancí speciálních struktur (*memory block*), které ukládají data meshpartu a do $n-1$ z nich, počínaje druhým, jsou deserializovány data. První se přeskočí, protože ten slouží jako defaultní buffer, do kterého se vždy deserializují ostatní meshparty, defacto znehodnocující vždy data předchozího meshpartu, který žádal o deserializaci a nebyl již v systémové paměti. Obrázek 5 zobrazuje hrubou podobu tohoto procesu v situaci, kdy jsou serializované tři meshparty (*cache file*) a pouze dva meshparty se vejdou do systémové paměti (*memory block*). Navíc je zde tzv. *assoc map*, která o každém meshpartu říká, ve kterém *memory blocku* je deserializovaný. Při indexu -1 není příslušný meshpart v paměti a pokud se chce deserializovat, musí se tak do defaultního *memory blocku* s indexem 0. Díky tomuto způsobu, i při více žádostech o deserializaci stejného meshpartu, nedochází k neustálému načítání z disku, jakmile se poprvé načte, v *assoc mapě* se pozmění vlastník defaultního *memory blocku* a při dalších dotazech se okamžitě vrací data tohoto *memory blocku*.



Obrázek 5: Schéma cache systému

Cache systém umí také ukládat výsledky metody konečných prvků, které jsou načteny ve své hrubé podobě a zde jsou rozdělena opět do jednotlivých částí, jeden pro každý meshpart a to tak, aby například pátému vertexu ze třetího meshpartu odpovídal opět pátý výsledek ze třetího meshpartu. Toho je dosáhnuto přes přepisovací tabulku vytvářenou během nahrávání modelu a poté uloženou na disk na konci serializace. Opětovná deserializace je samozřejmostí, ikdyž je potřeba pouze na začátku, při aplikaci nějakého výsledku metody konečných prvků na model, kdy se deserializuje a namapuje na texturové souřadnice způsobem popisovaným v sekci 2.2.

3.2.2 Serializace

Serilizace je proces uložení dat (v tomto případě dat meshpartů) na disk. Vzhledem k povaze aplikace, lze serializovat data pouze při načtení modelu. Vyjímkou jsou aktualizace dat, které ale neovlivňují celkovou velikost dat. Tento proces se zahájí zavoláním metody BeginSerializationProcess, inicializující hlavně použité buffery. Poté každý požadavek o serializaci uloží data meshpartu na disk. Ukládá se v binání formě. Problém nastává ale, jak uložit pointery, které obsahuje struktura ElementInfo na své indexy geometrií. Data, na která ukazuje daný pointer, nelze uložit ihned místo pointeru, jak by se to dělalo, pokud by se použily například streamy, jelikož by tím značně utrpěla rychlost aplikace, kdy pro každý ElementInfo by se muselo alokovat pole pro jistý počet indexů daný typem elementu a ty pak za sebou přečíst. Dynamické alokaci paměti při každé deserializaci lze předejít použitím vlastních tzv. file indexů místo pointerů, udávající offset příslušných dat v bajtech od počátku meshpartu, a tyto data, skrývající se za pointery, pak uložit v poli za sebou až po uložení všech ElementInfo. Při deserializaci pak načteme všechna data meshpartu najednou a jediná navíc operace je upravení file indexů, které se v ten moment nacházejí na místech pointerů, na skutečné pointery jednoduchým přičtením file indexu od počátku pointeru dat meshpartu. Listing 7 pak ukazuje jak tato serializace probíhá v praxi.

```
serialize (meshpart_data):
  if meshpart is empty then return
  start_offset = get actual position in file
  append array meshpart_data.vertices to file
  append array meshpart_data.normals to file
  # meshpart_data.geometry are indices to vertices
  foreach i as geometry flag combination in meshpart_data.geometry:
    append array meshpart_data.geometry[i] to file
  # meshpart_data.geometryIndex are indices to GeometryInfo array
  foreach i as geometry flag combination in meshpart_data.geometryIndex:
    append array meshpart_data.geometryIndex[i] to file
  # meshpart_data.gi are GeometryInfo instances in array
  append array meshpart_data.gi to file
  # calculating offset from beggining of data to the first indices
  # behind the array of ElementInfo
  ei_data_offset = get actual position in file
  ei_data_offset −= start_offset
  ei_data_offset += meshpart_data.ei.size ∗ (sizeof ElementInfo in bytes)
  # meshpart_data.ei are ElementInfo instances in array
```

```
# instead of pointer, offset is written
foreach ei in meshpart_data.ei:
  append ei.ret to file
  append ei_data_offset to file
  ei_data_offset += (nb of geometries for ei) * 4
# ElementInfo indices are written last one after another
foreach ei in meshpart_data.ei:
  append array ei.g to file
```

Výpis 7: Proces serializace dat meshpartu

Jakmile je poslední meshpart serializován, musí se zavolat metoda EndSerializationProcess. Jejím úkolem je vypočítat kolik maximálně meshpartů lze uchovat v systémové paměti a vytvoří příslušný buffer. Následně se do něj deserializuje $n - 1$ meshpartů, jak již bylo popisováno v předchozí sekci 3.2.1. Defaultní buffer pro zbývající meshparty je dimenzován na takovou velikost, aby se do něj vešel jakýkoliv, dosud nedeserializovaný meshpart. Během tohoto procesu se také automaticky aktualizuje *assoc map*, abychom mohli zpětně získávat informace o vlastníkovi meshpartu jednotlivých bufferů.

Při běhu aplikace, kdy uživatel může mírně upravovat model, je nezbytné tyto změny opět aplikovat i na data na disku. Může jít například o selekci geometrií, která pozměnuje vlastnosti GeometryInfo struktur. Proto je k dispozici speciální metoda, mimo serializační část, která tyto změny provede i na disku.

3.2.3 Deserializace

Princip opačné operace k serializaci již byl mírně vysvětlen při popisu serializace. Pokud meshpart zažádá o deserializaci, nejdříve se zkontroluje *assoc map* na indexu meshpartu a pokud se zde nachází validní index do pole *memory bloků*, pak jsou ihned vrácena jeho data v paměti. V opačném případě se změní aktuální vlastník defaultního bufferu na ten, který zrovna žádal o deserializaci a do bufferu jsou z disku nahrána jeho data. Postup tohoto nahrávání je znázorněn v Listing 8.

```
deserialize(meshpart_data):
  if meshpart is empty then return
  offset = offset of given meshpart from beggining of file in bytes
  read array meshpart_data.vertices from file at offset
  offset += meshpart_data.vertices size in bytes
  read array meshpart_data.normals from file at offset
  offset += meshpart_data.normals size in bytes
  # meshpart_data.geometry are indices to vertices
  foreach i as geometry flag combination in meshpart_data.geometry:
    read array meshpart_data.geometry[i] from file at offset
    offset += meshpart_data.geometry[i] size in bytes
  # meshpart_data.geometryIndex are indices to GeometryInfo array
  foreach i as geometry flag combination in meshpart_data.geometryIndex:
    read array meshpart_data.geometryIndex[i] from file at offset
    offset += meshpart_data.geometryIndex[i] size in bytes
  # meshpart_data.gi are GeometryInfo instances in array
  read array meshpart_data.gi from file at offset
  offset += meshpart_data.gi size in bytes
```

```
# meshpart_data.ei are ElementInfo instances in array
# instead of pointer, offset is written
read array meshpart_data.ei from file at offset
offset += meshpart_data.ei size in bytes
# ei.g is offset from beginning of meshpart data
foreach ei in meshpart_data.ei:
  ei.g = meshpart_data pointer + (int)ei.g
```

Výpis 8: Proces deserializace dat meshpartu

3.3 Memory Manager

Při počátečním načítaní modelu se používá několik dočasných bufferů, většinou pro data, která jsou z počátku pro celý mesh a jestě nerozdělené do jednotlivých meshpartů. Za normálních okolností s tímto není problém a můžeme jednoduše alokovat jeden objekt za druhým a přidat do bufferu. Bohužel v tomto případě, alokace geometrií a elementů po jednom velmi uškodí rychlosti nahrávní modelu a to nemluvě o dealokaci poté, co tyto buffery už nejsou potřeba, která je podstatně náročnější. Při dynamické alokaci na haldu stačí najít dostatečně velké volné místo a zabrat jej, pro dealokaci z haldy se už ale musí spojit okolní místa do jednoho a znovu vyvážit graf. Pokud k tomuto musí dojít např. tisíc-krát, tak uvolnění bufferů trvá déle než samotné nahrávání modelu. Z tohoto důvodu jsem napsal jednoduchý správce paměti. Pro každý typ objektu existuje samostatný správce díky c++ šablonám a pro snadnou integraci do existující implementace jsem napsal několik maker (viz. seznam 9)

```
// equivalent of 'new GeometryInfo()',
// calls default constructor
#define NEW_GEOMETRYINFO0() \
  SimpleMemMgr<GeometryInfo>::Instance().Alloc(GeometryInfo())
// equivalent of 'new GeometryInfo(copy)',
// calls copy constructor
#define NEW_GEOMETRYINFO1(copy) \
  SimpleMemMgr<GeometryInfo>::Instance().Alloc(copy)
// equivalent of 'delete ptr', calls destructor
#define DEL_GEOMETRYINFO(ptr) \
  SimpleMemMgr<GeometryInfo>::Instance().Free(ptr)

// equivalent of 'new ElementInfo(ret, domain)',
// calls constructor with two parameters
#define NEW_ELEMENTINFO0(ret, domain) \
  SimpleMemMgr<ElementInfo>::Instance().Alloc(ElementInfo(ret, domain))
// equivalent of 'delete ptr', calls destructor
#define DEL_ELEMENTINFO(ptr) \
  SimpleMemMgr<ElementInfo>::Instance().Free(ptr)
```

Výpis 9: Makra a použité typy objektů v programu

Správce paměti používá, předem definované, stejně velké bloky paměti, ze kterých vždy alokuje objekty. Při každém požadavku na alokaci se nejdříve najde blok paměti, který má volné místo a zabere místo na jeden objekt. Pokud nenajde volný blok paměti

s místem, tak vytvoří další blok. Každý blok paměti je alokován pomocí fce malloc(), která ale nepracuje s třídami a nevolá žádné konstruktory. Toto chování je ale nezbytné, kdyby totiž volal konstruktory, tak by při každém vytvoření nového bloku vytvořil velké množství objektů, které ani nemusí být potřeba a hlavně by volal pouze defaultní konstruktory, ikdyž bychom potřebovali volat jiné. Proto samotné volání konstruktoru je až při požadavku na alokaci a to přes tzv. placement new operator:

```
Foo* p;
// this code...
p = new Foo();
// ... is equivalent with
void* raw = malloc(sizeof(Foo));
try
{
  p = new(raw) Foo();
}
catch (...)
{
  p->~Foo();
  throw;
}
```

Výpis 10: Placement new operator

Uvolnění paměti je odkládáno až do doby, kdy v bloku již není žádný alokovaný objekt a teprve poté se uvolní celý blok.

Díky tomuto postupu se alokace a hlavně dealokace paměti velmi výrazně zlepšila a i pro velmi velké objekty s mnoha geometriemi a elementy netrvá finální dealokace bufferů déle než 2 sekundy oproti původní verzi bez nějak speciální správy paměti, kde ten čas už byl téměř minuta.

3.4 Řezy rovinou

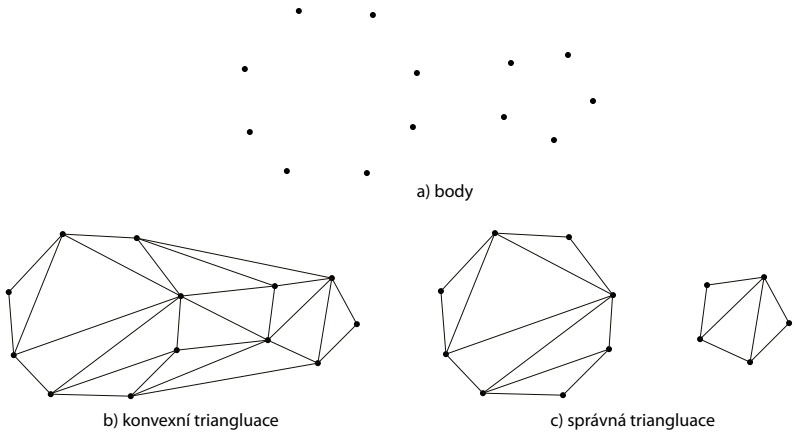
V této práci byl proces řezání objektu rovinou velmi důležitý a to nejen pro prohlížení objektu zevnitř, ale také pro vizualizaci volumetrickou metodou. Samotný proces řezání by ale bez nějaké pomocné datové struktury byl velmi pomalý. Každá geometrie by se musela testovat na průsečík s rovinou. Existují ale speciální datové struktury, díky kterým lze některé skupiny geometrií úplně vynechat z výpočtů a dosáhnout tak lepších výpočetních časů. A o nich je následující sekce.

3.4.1 Pomocná datová struktura

Pokud bychom chtěli efektivně pracovat nad velkými daty, jako například v tomto případě vyhledávání průsečíků modelu s obecnou rovinou, v lepším než lineárním čase, je zapotřebí využít některou z datových struktur pro rozdělení prostoru na části. K výběru jsou struktury jako kd-tree, octree, bsp-tree, R-tree a další [Sam88].

Pro svou práci práci jsem zvolil octree, jelikož velmi šikovně popisuje rozdělení prostoru a je šetrnější k využité paměti než například kd-tree, který by vyžadoval na každou

geometrii jeden node ve stromě, a ten dva ukazatele na své dva poloprostory a index pro indentifikaci geometrie. Pro řezy rovinou a následnou triangulaci bodů v rovině řezů je mnohem výhodnější pracovat s elementy než-li s geometriemi, jelikož pak lze řezat po jednotlivých elementech a ihned poté ztriangulovat jen velmi málo bodů, pro což není potřeba žádný univerzální algoritmus, ale stačí jej na pevno naprogramovat pro několik málo variant. Pokud by se řezalo po geometriích, vznikalo by velké množství bodů, které by se muselo rychle ztriangulovat a aby to nebylo tak snadné, tak výsledkem nemusí být vždy nějaký konvexní útvar jak ukazuje Obrázek 6, kde se řezaly dva válce. Proto by téměř nebylo možné získat správný výsledek v nějakém rozumném čase. kd-tree (ale i některé další datové struktury) by s elementy měli problémy, jelikož elementy nerozdělují prostor přesně na dva poloprostory a muselo by se počítat s nějakou umělou rovinou elementu, která by efektivní práci velmi ztížila.



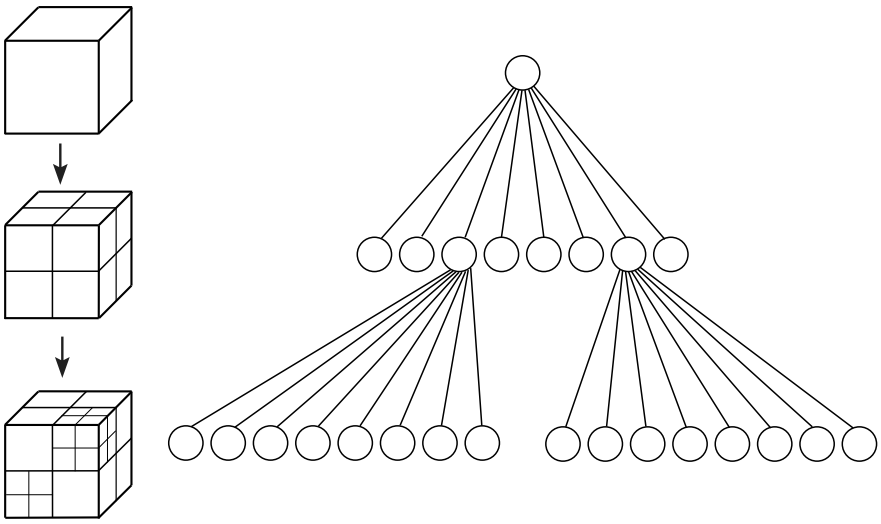
Obrázek 6: Triangulace dvou kružnic. a) Vstupní body b) Triangulace vzniklá použitím standartních triangulovacích technik (např. Delaunay triangulace) c) Správně triangulované dvě kružnice

S velkým množstvím elementů by kd-tree i naplno otestoval rychlost systémové správy paměti a to hlavně při dealokaci celého stromu, která by mohla trvat i několik sekund. Přibližné srovnání pamětových nároků různých datových struktur je v Table 7. Celá struktura octree ovšem i tak zabírá nemalé množství paměti a pokud by to byl problém, řešit by se mohl cachováním na disk [Tia03].

	Počet elementů				
	1000	5 000	20 000	100 000	1 000 000
kd-tree	24kB	120kB	480kB	2.4MB	24MB
octree	4.4kB	21.8kB	85kB	0.12MB	4.2MB

Obrázek 7: Přibližné porovnání pamětových nároků datových struktur

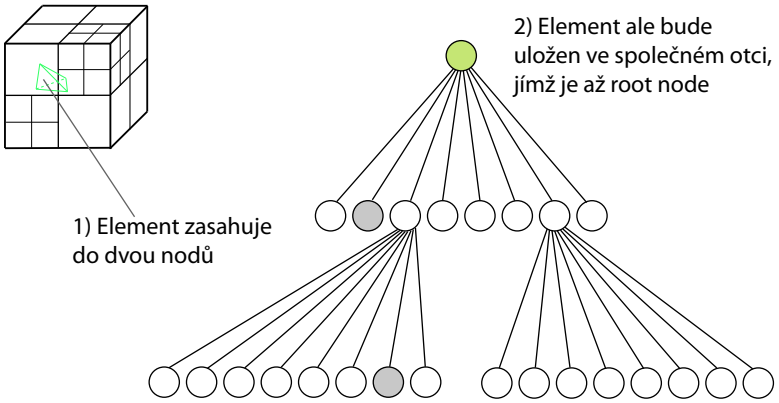
Octree neboli tzv. oktantový strom je stromová datová struktura, ve které každý vnitřní node má až 8 potomků. Používá se na rekurzivní rozdělování 3D prostoru na menší podprostory s cílem uzavírat blízké objekty do stejných podprostorů. Vzhled takovéto struktury je na Obrázek 8. Často se používá pro prostoré indexování, view frustum culling, detekce kolizí a podobně.



Obrázek 8: Octree - vzhled datové struktury

Hlavní výhodou této struktury je, že pokud jeden node splňuje nějakou podmínku (např. celý se nachází uvnitř view frustum kamery nebo v mém případě je celý nad řeznou rovinou) lze ihned říct, že všichni jeho potomci splňují tuto podmínku také a nemusí se vůbec na tuto podmínku také testovat. Navíc každý node může obsahovat nějaké, předem zvolené, maximum objektů, ne pouze jeden, jako tomu je u například kd-tree. Jednoznačné zrychlení je proto zřejmé, kdy místo lineární časové složitosti, pokud by se nepoužila žádná speciální struktura a všechny objekty by se testovali za sebou, získáme čas logaritmický, protože pokud jeden node nesplňuje podmínku, lze říct, že všichni jeho potomci tu podmínku také nesplňují a jediný průchod stromem je, pokud je podmínka částečně splněna (například rovina řeže node). V takovém případě se musí rekurzivně vstoupit do tohoto node a testovat podmínku na jeho potomcích. Pokud by se jako objekty octree používali body, byl by toto způsob jak procházet stromem. V tomto případě ale jako objekty octree slouží elementy, které samy zabírají nějaký prostor a mohou tak zasahovat do více nodů současně. Tento problém je vyřešen tak, že pokud by musel být element ve více než v jednom nodu současně, tak bude patřit společnému otci těchto nodů. To slovo "společnému"je důležité, jelikož element může patřit do nodů i z jiných úrovní zanoření ve stromě, jak ukazuje Obrázek 9.

Celkový proces přidávání elementů do octree shrnuje state diagram v Obrázek 10.

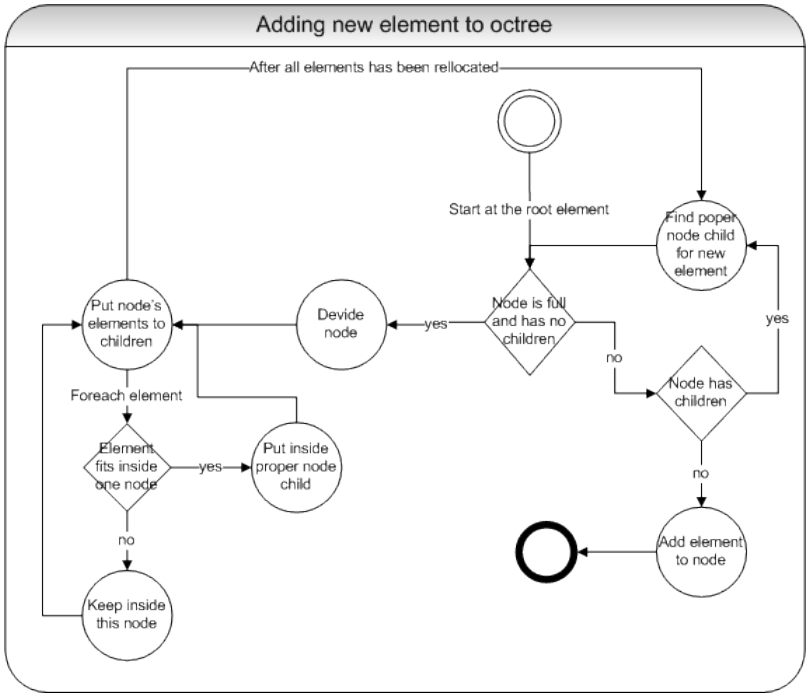


Obrázek 9: Element zasahující do více nodů

Před tím, než lze vkládat nové elementy, musí se vytvořit jeden hlavní node box s takovými rozměry, aby obaloval všechny elementy (prakticky se jedná o tzv. axis-aligned bounding box, AABB) a stanoví se hodnota n jako maximální množství elementů, které smí být v jednom nodu. Tato hodnota by měla růst logaritmicky s počtem elementů vzhledem k povaze stromů jako datových struktur. Přesněji už ovšem lze jen velmi obtížně vyjádřit a tak jsem po několika experimentech stanovil hodnotu n podle Equation 6. V Obrázek 11 je pak vidět růst této hodnoty.

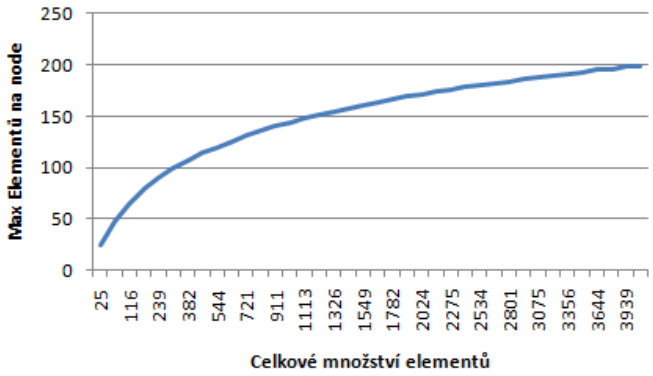
$$n = max(1.0, log(1.0 + NbOfElements/28.0) * 40.0) \tag{6}$$

Poté lze začít s přidáváním elementů. Prvních n elementů se jednoduše přidá do root nodu, ale s n+1 elementem se node rozdělí na osm stejných poloviční boxů a všechny elementy patřící do tohoto děleného nodu se musí znovu rozmístit do těchto osmi nových nodů. To je naštěstí velmi snadná a rychlá operace, jelikož výpočet indexu boxu, do kterého náleží jeden bod ukazuje Listing 11 a pro názornost i Obrázek 12 . U elementů se vypočítají indexy boxů pro všechny jeho body stejným vzorcem s tím, že pokud nejsou všechny indexy shodné, tak element náleží do více nodů, a proto zůstane ve stejném nodu. Pokud ovšem jsou všechny indexy stejné, tak můžeme element přímo vložit do nodu s daným indexem a to i bez jakýchkoliv kontrol, jelikož při dělení nodu má tento node n elementů a všech n elementů může bez problému jít do stejného potomka. Problém pak nastává s elementem, který nově přidáváme, a to pokud všechny tyto elementy šli do stejného potomka a tento nový element tam patří také. Z tohoto důvodu se tento nový element nepřidává prostým přidáním do pole, ale znovu zavoláním vkládací metody nad potomkem. Na první pohled se může zdát, že tento krok se může teoreticky nekonečně zacyklit, ale pravdou je, že s každým dělením nodu, mají jeho potomci poloviční velikost a tak je čím dále obtížnější vměstnat všechny elementy vždy pouze do jednoho, neustále se zmenšujícího, boxu. Metoda navíc může vracet false, pokud už následné dělení nemá smysl, tedy stav, kdy všechny elementy daného nodu již nelze



Obrázek 10: Vkládání nového elementu do octree

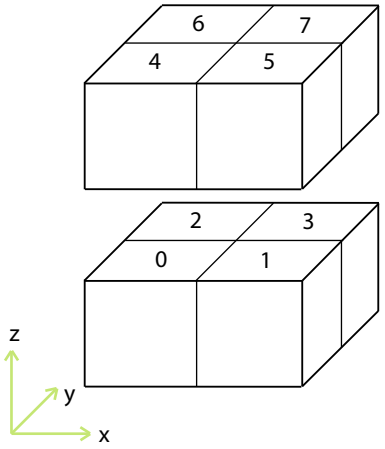
vložit do žádného potomka, a předchozí volání metody pak na tuto skutečnost reaguje tak, že jej zkusí vložit k sobě. Pokud pro něj má místo, pak je všechno v pořádku a element se jednoduše přidá, v opačném případě metoda opět musí vrátit false a to tak hluboko, dokud se nenajde pro něj místo nebo nedojde na root node. Pokud i on je již plný, tak se nový element ukládá ve speciálním poli. Tato část bohužel zhoršuje efektivnost octree, ale na druhou stranu se nejedná o příliš velké počty, které by měly nějaký zásadnější vliv.



Obrázek 11: Růst maximálního počtu elementů na každý node v závislosti na celkovém počtu elementů

```
index = 0;
index |= bod.x >= node_center.x ? 1 : 0;
index |= bod.y >= node_center.y ? 2 : 0;
index |= bod.z >= node_center.z ? 4 : 0;
```

Výpis 11: Výpočet indexu potomka v závislosti na ose souřadnic, pozici bodu a středu nodu



Obrázek 12: Indexy potomků jednoho nodu v závislosti na ose souřadnic

Jak vypadá ale tato datová struktura octree naimplementovaná si lze povšimnout v Listing 12. Výsledek každého řezu se vždy ukládá do několika dynamických polí reprezentovaných třídou CDynArray (velmi podobná struktura k std::vector, ale o poznání rychlejší). Geometrie, které budou ve výsledku celé, se ukládají jako indexy na ně do pole

m_Clipped a to podle typu geometrie, úsečky se ukládají do m_Clipped[0], trojúhelníky do m_Clipped[1] a čtyřúhelníky do m_Clipped[2]. Stejný princip výběru indexu pole se používá i u těch zbývajících polí. Geometrie, které protnuly řeznou rovinu a ty nově vzniklé, na úrovni roviny aby zaceli díru, se ukládají do m_ClippedNewVerts, m_ClippedNewNormals a m_ClippedNewResults. Tentokrát již ale ne jako indexy do původních polí s body, normálami a výsledkami konečných prvků, ale ukládají se přímo hodnoty a pracuje se s nimi tak bez indexů. Platí, že například pro trojúhelníky budou každé tři hodnoty tvořit jeden trojúhelník a podobně.

```
class Octree
{
private:
  BoundingBox m_AABB;
  SNode*      m_pRoot;
  // elements that didn't fit inside the octree, must be processed as well
  CDynArray<int> m_RemainingTooBigElms;
  // clipping & slicing results
  // indices for geometries that are completely in positive half-space
  mutable CDynArray<int> m_Clipped[3];
  // clipped new points, for GFT_TRIANGLE ([1]) every three vertices
  // creates triangle, ...
  mutable CDynArray<Point> m_ClippedNewVerts[3];
  // clipped new normals
  mutable CDynArray<Vector3D> m_ClippedNewNormals[3];
  // clipped new results
  mutable CDynArray<RESULTDATA_TYPE> m_ClippedNewResults[3];

  // called everytime some gemetryinfo falls entirely into positive half-space
  void Hit_PlanePositiveHalfSpace(const GeometryInfo* pGeom) const;
  // called when line intersects plane. intersect point is
  //  first_point + (second_point - first_point ) * pdEdgeCoords[0]
  // pHalfSpace says foreach point if it's in positive half-space and uNbOfPositives is
  // just to say absolute number of those points in positive half-space
  void Hit_Plane_Line(const GeometryInfo* pGeom, const f64* pdEdgeCoords,
    const s32* pHalfSpace, u32 uNbOfPositives) const;
  // analogy to Hit_Plane_Line. Except that there are usually two intersect points
  // and pdEdgeCoords[0] is interpolation coef from first point to second,
  // pdEdgeCoords[1] from second to third and pdEdgeCoords[2] from third to first
  void Hit_Plane_Triangle(const GeometryInfo* pGeom, const f64* pdEdgeCoords,
    const s32* pHalfSpace, u32 uNbOfPositives) const;
  // analogy to Hit_Plane_Triangle, except of obvious change of geometry type
  void Hit_Plane_Quad(const GeometryInfo* pGeom, const f64* pdEdgeCoords,
    const s32* pHalfSpace, u32 uNbOfPositives) const;
public:
  OctTree();
  ~OctTree();
  // creates octree from array of elements. Also there has to be array
  // of points a geometryInfo associated with current meshpart
  void BuildFromArray(const Point* pVerts, const ElementInfo* pElms,
        const GeometryInfo* pGeoms, u32 unElmCount,
        const BoundingBox& AABB);
  // clipping, used for direct rendering
```

```
  void Clipping_Calc(const Plane& plane, const Point* pVertices,
        const Vector3D* pNormals, const ElementInfo* pElms,
        const GeometryInfo* pGeoms, const RESULTDATA_TYPE* pResult,
        u32 uResultDimension, bool bFastOne) const;
  const V_Index* Clipping_GetGeomIndices(GeometryFlag_Type eType) const;
  u32 Clipping_GetNbOfIndices(GeometryFlag_Type eType) const;
  void Clipping_GetNewGeoms(const CDynArray<Point>** ppPositions,
        const CDynArray<Vector3D>** ppNormals,
        const CDynArray<RESULTDATA_TYPE>** ppResults) const;
  // slicing, used for volumetric rendering
  void Slice_Calc(const Plane& plane, const Point* pVertices,
        const ElementInfo* pElms, const GeometryInfo* pGeoms,
        const RESULTDATA_TYPE* pResult, u32 uResultDimension) const;
  void Slice_GetResult(const CDynArray<Point>*& pPositions,
        const CDynArray<RESULTDATA_TYPE>*& pResults) const;
};
```

Výpis 12: Rozhraní a atributy oktantového stromu

Jednotlivé nody octree jsou reprezentovány následující třídou Listing 13. Aby se vyhnulo neustálému předávání mnoha parametrů, jako jsou ukazatele na pole bodů, normál, výsledků konečných prvků, geometrií, elementů atd., tak jsou tyto data uložena do statických atributů této třídy. Při každém požadavku na výpočet řezu jsou přiřazena odpovídající data a po skončení opět vynulována, aby nedošlo k nějakým milným přístupům do paměti, když data, na která atributy ukazují, mohou být i pouze dočasná.

Každý node si ukládá pole indexů do všech elementů, které mu jsou přiřazeny, množství těchto elementů a pole ukazatelů na jeho osm potomků. Kromě toho poskytuje mnoho metod pro řezy, které jsou posány v sekcích 3.4.2 a 3.4.3.

```
class Node
{
  // i've chosen to set many variables as static rather then
  // constantly passing them as parameters to every method
  static u32                    s_uMaxFacesPerNode;
  static const ElementInfo*     s_pFirstElm;
  static const GeometryInfo*    s_pFirstGeom;
  static const Point*           s_pPoints;
  static const Vector3D*        s_pNormals;
  static Plane                  s_Plane;
  static const Octree*          s_pOwner;
  static const RESULTDATA_TYPE* s_pResult;
  static u32                    s_uResultDimension;

  // number of elements in node
  s32       nElmsCount;
  // array of all element indices which belongs to this node
  El_Index* pElms;
  // 8 children, for list node it's 8 nulls
  Node*     pChildren[8];

  Node();
  ~Node();
  // creates array of elements if necessary
```

```
  void InitElmArray();
  // releases array of elements if present
  void ReleaseElmArray();
  // to save some memory, each node's position and dimensions are
  // fastly calculated by this method from parent parameters
  static void GetBoxFromParentStats(const Point& ptParentCenter,
    const Point& ptParentDim, u32 unChildID, Point& ptOutCenter);
  // simply add new element index
  void EasyInsert(EI_Index ElmI);
  // tries to add new element index, if node is
  // already full then return false
  bool TryToInsert(EI_Index ElmI);
  // recursivelly find proper node for new element and adds it
  bool Insert(const ElementInfo* pNewElm, const Point& ptCenter,
    const Point* ptHalfDim);
  // for visualization clipping, the fast one
  void Clipping_Simple(const Point* pCenter,
    const Point* pDimHalf) const;
  // for visualization clipping, the full one
  void Clipping_Full(const Point* pCenter,
    const Point* pDimHalf) const;
  // adds all elements of this and all children to clipping result
  void Clipping_AddEverything() const;
  // for volumetric rendering, calc one slice
  void Slice_Calc(const Point* pCenter, const Point* pDimHalf) const;
  // triangulates set of points and adds them to clipping result
  static void Triangulate(const Point* pts, const RESULTDATA_TYPE* res,
    u32 uNbOfPts);
  static void Triangulate(const Point* pts, const RESULTDATA_TYPE* res,
    u32 uNbOfPts, const Vector3D& vNorm);
  // processing elements and calculating new clipped geometries
  // if needed
  static void Clipping_ProcessElement_Full(EI_Index i);
  static void Clipping_ProcessElement_Simple(EI_Index i);
  static void Slice_ProcessElement(EI_Index);
  // to unify adding results with different dimensions
  // simple add from vertex index
  static void AddResult(GeometryFlag_Type g, V_Index a);
  // add interpolated value
  static void AddResult(GeometryFlag_Type g,
    V_Index a, V_Index b, f64 t);
  // simple add
  static void AddResult(GeometryFlag_Type g,
    const RESULTDATA_TYPE* p);
  // returns number of elements in this node and his children
  u32 GetCubeElms() const;
};
```

Výpis 13: Rozhraní a atributy nodu oktantového stromu

3.4.2 Řezy pro přímou vizualizaci

Jedním z požadavků na aplikaci bylo, aby se model mohl řezat rovinou s cílem zobrazit pouze jednu část prostoru. Právě kvůli tomuto účelu je potřeba octree, přes který je tento výpočet mnohem rychlejší. Po aktivování řezání se tedy vybuduje oktantový strom pro každý meshpart. Implementovány jsou dvě varianty řezů pro přímou vizualizaci.

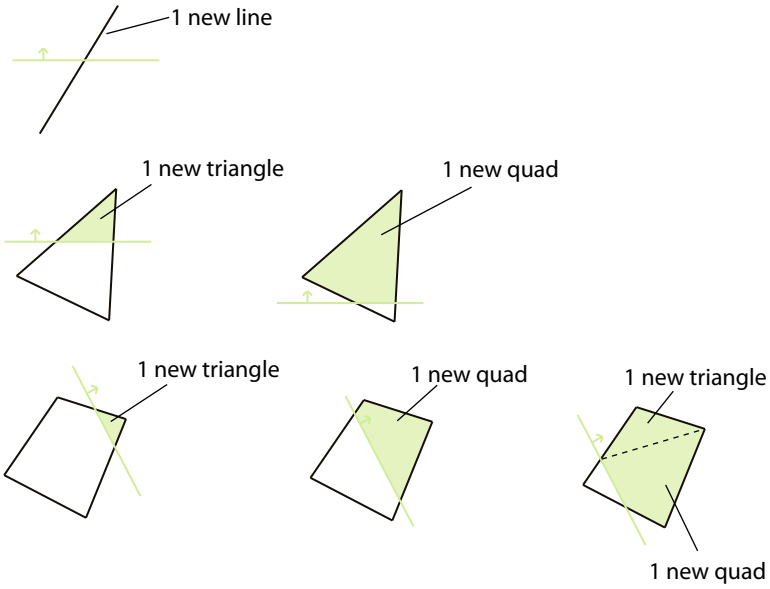
- *fast*: Tato metoda se používá, když se řezná rovina animuje, aby se získala co nejpříznivější odezva aplikace. Pro vizualizaci se používají pouze ty elementy, které celé patří do kladného poloprostoru od roviny řezu. Ve výsledku je pak hranice řezu modelu velmi členitá a ne příliš přesná, navíc kvůli použitým vnitřním normálám zde není ani správně vypočítané stínování. Nevýhody této metody se ale ztrácí ve faktu, že výpočet řezu je podstatně rychlejší. V aplikaci je tato technika implementována metodou Octree::Clipping_Calc s parametrem bFastOne = true

- *precise*: Jakmile se animace pohybu řezné roviny dokončí, zavolá se tato metoda, která vypočítá už přesný model. V zásadě se jedná o metodu *fast* doplněnou o zpracování elementů, které rovina řezu protne. V aplikaci je tato technika implementována metodou Octree::Clipping_Calc s parametrem bFastOne = false

Oboje metody začínají na root nodu a testují všechny potomky na protnutí s rovinou řezů a rekurzivně volají metodu (v implementaci jde o metody Octree::Node::Clipping_Simple pro *fast* a Octree::Node::Clipping_Full pro *precise* techniky) na těch potomcích, které protnou. Výsledky těchto metod jsou pak uloženy ve speciálních polích, v závislosti na tom, zda geometrie patří celá do výsledku nebo jen jeho část a podle typu geometrie. Nody, které jsou celé v kladném poloprostoru řezné roviny, jsou automaticky zařazeny do výsledného modelu a to tak, že se také rekurzivně procházejí už ale jinou metodou nazvanou Octree::Node::Clipping_AddEverything, která přídá všechny elementy vlastněny daným nodem a zavolá stejnou metodu na všech svých potomcích a tak se efektivně přidají všechny elementy obsaženy v nodu, na kterém se tato metoda napočátku zavolala. Nody v záporném poloprostoru jsou ihned vynechány z dalších výpočtů. Nakonec se musí i lineárně projít všechny elementy ze speciálního pole, popsaném v sekci o přidávání elementů do octree (v Listing 12 je to atribut Octree::m_RemainingTooBigElms).

Pro všechny elementy protnutých nodů se u *precise* varianty musí volat metoda, která spočítá protnutí elementu s rovinou, vypočítá nově vzniklé geometrie, včetně připadných výsledků metody konečných prvků a vertex normál, a vloží je do výsledku řezu. Tyto nové geometrie vznikají ze dvou různých cest.

- *Ořezaná původní geometrie* - pokud geometrii protne řezná rovina, zachová se z ní pouze část a v závislosti na místě protnutí, pak například u trojúhelníku může vzniknout jiný menší trojúhelník nebo čtyřúhelník. Ostatně všechny možné varianty jsou zobrazeny v Obrázek **??**. Jejich výpočet začíná zjištěním o kterou varinatu se přesně jedná a poté pro každý tento případ se zjistí, které body jsou v kladném a které v záporném poloprostoru. S těmito informacemi jde pak pouze o výpočet nových bodů na rovině řezu a vytvoření nové geometrie.



Obrázek 13: Varianty ořezaných původních geometrií, světle zeleně je zobrazena řezná rovina

- *Geometrie ležící na rovině řezu* - Tyto gemetrie vznikají na rovině řezu elementu. Jejich výpočet je o poznání složitější, než jak tomu bylo u předchozí varianty, ale je s ním částečně propojen. Nově vzniklé interpolované body na rovině řezu jsou přidávány do dočasného bufferu tak, aby ale neobsahoval duplikáty. Tomu se zabraňuje lineárním procházením celého bufferu bodů s testem na ekvivalenci bodu v bufferu s novým bodem. Díky velmi malým množstvím bodů (v praxi maximálně šest) není lineární průchod problémem. Jelikož operátor ekvivalence == u plovoucích čísel není přiliš přesný, díky nepřesné interpretaci takovýchto čísel, musí se počítat s jistou maximální možnou odchylkou. Poté, co se projdou všechny geometrie elementu, přijde na řadu triangulace všech bodů z dočasného bufferu. Všechny typy elementů jsou konvexní útvary, a proto všechny možné řezy jakýmkoliv elementem budou vždy tvořit opět 2D konvexní útvar. Tento fakt velmi usnadnil práci, hlavně pro triangulaci více jak čtyř bodů, u kterých už vzniká více jak jedna nová geometrie. Pro méně jak čtyři body je situace snadná, stačí body interpretovat jako jedna geometrie. Pouze u čtyřúhelníků se musí dát pozor na pořadí bodů. V ostatních situacích s více body se použije metoda Octree::Node::Triangulate. Výpočet se skládá ze tří fází.

  1. *Nalezení nejkrajnější hrany z prvního bodu* - Na počátku se na pevno stanoví hrana z prvního bodu do druhého a k ní se počítají úhly k ostatním hranám vycházejících z prvního bodu. Cílem této fáze je najít jednu okrajovou hranu a vyhnout se tak nějaké vnitřní hraně. Vzhledem ke konvexnosti lze tuto hranu

z prvního bodu vždy najít. Pokud by tato počáteční hrana již byla okrajová, tak tato fáze by pouze nalezla onu druhou okrajovou hranu vycházející ze stejného bodu.

2. *Výpočet úhlů mezi nejkrajnější hranou a ostatními hranami od prvního bodu* - Jakmile je známa jedna okrajová hrana, vypočítají se úhly mezi ní a ostatními hranami také vycházejících z prvního bodu ke všem ostatním.
3. *Procházení přes nejbližší hrany a přidávání geometrií* - Poslední fáze již vytváří jednotlivé geometrie. Snahou je vytvářet hlavně čtyřúhelníky, které vyžadují menší množství paměti nežli ostatní geometrie. Proto výstupem je vždy několik čtyřúhelníků a jeden nebo žádný trojúhelník. Začne se procházením podle úhlů od nejmenších k největšímu a s každou druhou hranou vzniká jeden čtyřúhelník. Pokud nezbyde hrana i pro ten poslední čtyřúhelník, je místo něj vytvořen trojúhelník.

Jak tento proces vypadá na obrázku se můžete podívat na Obrázek 14. Přibližná časová složitost této triangulace je $O(f(n)) = 3n^2$. Známé jsou sice algoritmy s lepšími časy, ale protože třídy složitostí nejsou příliš přesné a platí až od nějaké, blíže neznámé, velikosti problému a schovávájí konstanty, tak tento algoritmus bude pro tak malé množství bodů, pro které algoritmus navrhnut, o poznání rychlejší.

Na Obrázek 15 jsou zobrazeny všechny typy geometrií, na které je proces řezaní rozděluje. Jsou zde "Geometrie v kladném poloprostoru", které jsou ukládany do proměnné Octree::m_Clipped, "Úplně ořezané geometrie", které jsou vynechány a předchozí dva typy

3.4.3 Řezy pro volumetrické renderování

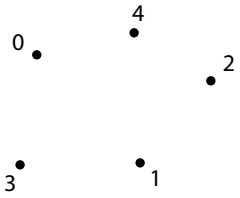
Volumetrické renderování vyžaduje velké množství paralelních řezů modelu. Vstupem této techniky jsou geometrie pouze v rovině řezu.

Způsob výpočtu řezů pro tento účel se liší jen velmi málo od toho původního, popsaného v předchozí sekci. Rozdíly se ale najdou. Metoda, která obstarává tento výpočet je Octree::Slicing_Calc a pracuje velmi podobně jako Octree::Clipping_Calc. Při průchodu nody stromu se elementy v kladném poloprostoru ale vynechávají stejně, jako ty v tom záporném. Pracuje se pouze s těmi, které rovina řízne. Geometrie protnuté rovinou jsou také ignorovány a používají se pouze interpolované body na rovině k následné triangulaci protnutého elementu. Defakto jsou výsledkem řezu pouze nově vzniklé geometrie na rovině řezu.

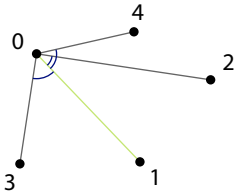
Požadavky na tyto řezy vznikají při inicializaci volumetrického rendereru, při prohlížení modelu již není tato metoda potřeba.

3.5 Architektura systému

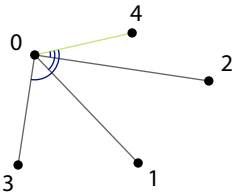
Aplikace využívá několik dalších knihoven pro realizaci celého programu. Použití konkrétních knihoven ale není nezbytné. Jako grafické API se mohlo použít i DirectX,



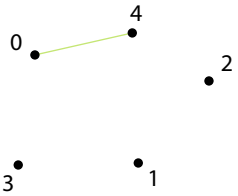
0) Vstupní body v náhodném rozložení



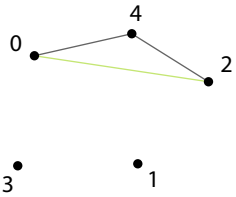
1) Od hrany [0,1] má největší úhel hrana [0,4]



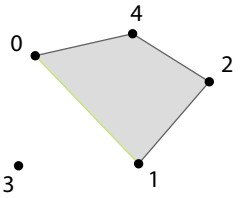
2) Úhly od hrany [0,4] k ostatním hranám



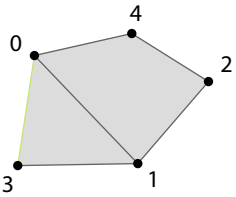
3.1) První hrana první geometrie



3.2) První trojúhelník, který ale lze převézt na čtyřúhelník

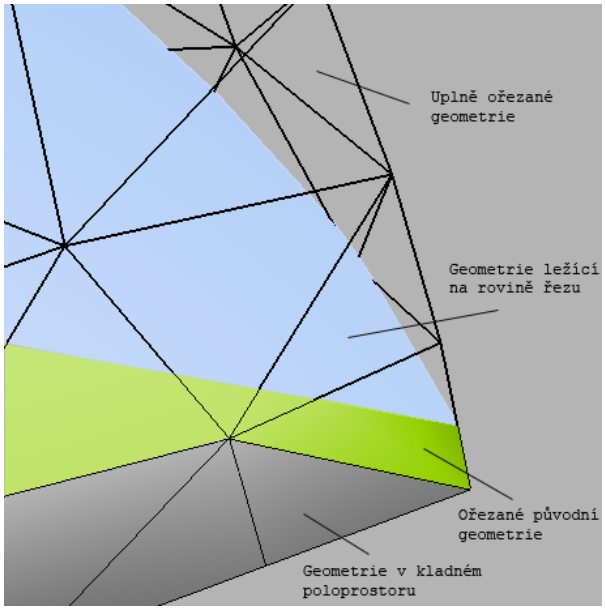


3.3) První čtyřúhelník, který se uloží s pořadím bodů [0,4,2,1]



3.4) Tento poslední trojúhelník se zapíše s pořadím bodů [0,1,3]

Obrázek 14: Postup při triangulaci bodů



Obrázek 15: Typy nově vzniklých geometrií při řezání rovinou

pokud by nebyla vyžadovaná multiplatformnost a GUI stejně tak mohlo být vytvářeno například přes knihovnu wxWidgets.

OpenGL - grafické API

OpenGL (Open Graphics Library) je průmyslový standard specifikující multiplatformní rozhraní (API) pro tvorbu aplikací počítačové grafiky [OpenGL]. Používá se při tvorbě počítačových her, CAD programů, aplikací virtuální reality či vědeckotechnické vizualizace apod. Veškerá činnost OpenGL se řídí vydáváním příkazů pomocí volání funkcí a procedur (kterých OpenGL definuje cca 250). V OpenGL se nepoužívá objektově orientované programování. V této aplikaci je OpenGL nezbytný pro samotný proces renderování rasterizací a volume renderováním. Selekce polygonů uživatelem je také hardwarově řešená přes toto rozhraní.

Qt - knihovna pro vytváření GUI

Qt je jedna z nejpopulárnějších multiplatformních knihoven pro vytváření programů s grafickým uživatelským rozhraním [Qt]. Při vytváření této aplikace byla použitá aktuální verze 4.6.2.

QGL Viewer - OpenGL widget do Qt

Tato knihovna usnadňuje propojeni OpenGL a Qt vytvořením speciálního widgetu, který v sobě již obsahuje veškerou potřebnou správu OpenGL a kamerový systém. Qt má zabudovanou podporu pro OpenGL a tak nebyla tato knihovna příliš nutná, ale i tak obsahuje některé navíc funkce oproti té původní z Qt, které dále usnadňují práci.

4 Benchmark

Pro určitou formu testování výkonu nejnáročnější části této práce, výpočty řezů modelu rovinou, byl napsán jednoduchý test rychlostí těchto výpočtů pro různé varianty. Do časů jsou zahrnuty i časy potřebné pro deserializace jednotlivých meshpartů. Jelikož jsou zahrnuty ale do všech testů, jedná se jen o malou konstantu a neovlivňuje tak výsledky ve větší míře.

Testovací prostředí

Pro teamovou spolupráci na aplikaci jsme využívali Microsoft Team Foundation Server pro správu verzí. Proto byla aplikace vyvíjená v prostředí Microsoft Visual Studio Team System 2008. Pro vyvíjení a testování sloužilo hardwarové prostředí v Table 1.

CPU	Intel Core 2 Quad Q6600
RAM	6 144MB
GPU	ATI Radeon HD 2900 XT
OS	Microsoft Windows 7

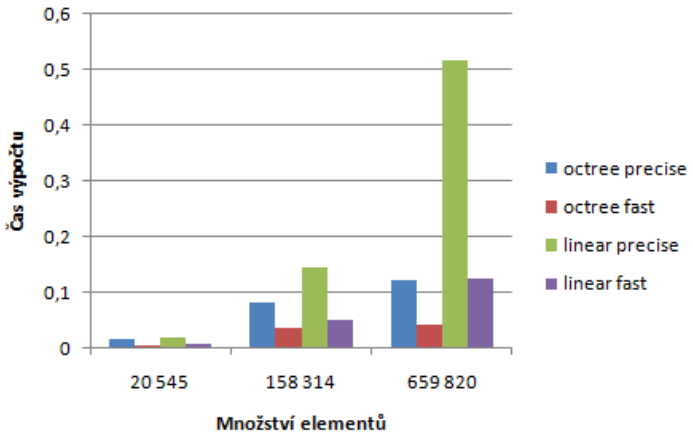
Tabulka 1: Použité testovací HW vybavení

Jednotlivé testy

Následují testy rychlostí výpočtu řezů, a to obou metod *fast* a *precise* počítané nad oktantovém stromu i nad lineárním poli. Pro testování posloužili tři různé modely, které v době implementace byly k dispozici a byly přiměřeně složité. U malých modelů se výkon počítá obtížně, jelikož u nich stoupá vliv chyby měření s klesající velikostí modelu. Všechny varianty výpočtu řezů se počítali vždy se stejnou rovinou řezu. Časy jsou měřené v sekundách.

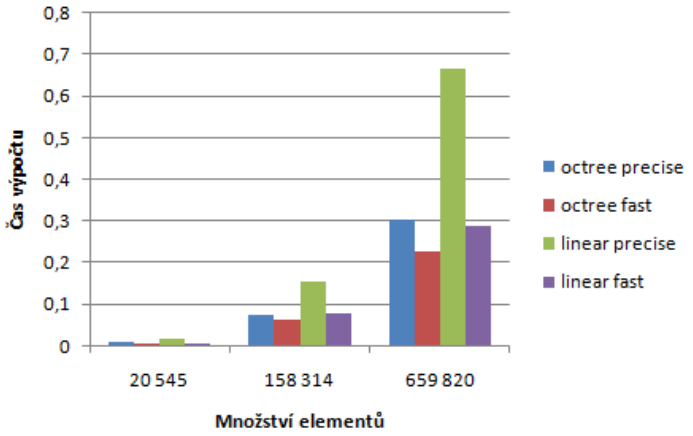
První test (Obrázek 16) je proveden s rovinou řezu protínající, pokud možno, co největší část modelu. V zásadě tato rovina prochází zhruba středem modelu. Je to test, který je nejvíce závislý na použitém algoritmu výpočtu chybějících a protnutých geometrií. Podle očekávání byla nejrychlejší metoda *fast* počítaná přes octree. O následující umístění spolu soupeří *precise* metoda přes octree a *fast* metoda nepočítaná nad žádnou speciální datovou strukturou. Způsobené to je právě faktem, že *fast* metoda nepočítá chybějící a protnuté geometrie v rovině řezu, kterých je v tomto testu co nejvíce. Octree varianta je ale lepší až pro větší modely.

Další test (Obrázek 17) patří mezi ty náročnější, které z modelu vyřežou pouze malou část, a tak zůstává téměř celý model. Zde oktantový strom příliš nepomůže, vzhledem k tomu, že skoro každá geometrie projde testem na řez rovinou a nejsou tak odstraněné velké části modelu z budoucích výpočtů. Na druhou stranu zase projde více boxů testem, a tak se sníží celkový počet nutných testů na průsečík u elementů. I tak jsou rozdíly mezi octree a lineárním řešením relativně malé. Vyjímkou je *precise* metoda u lineární varinaty,



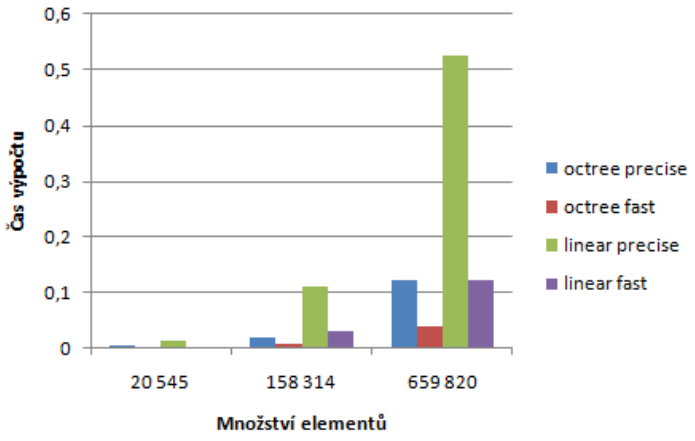
Obrázek 16: Graf výkonu řezu středem modelu

kde je volána složitější funkce na výpočet, které trvá déle, než zjistí, že geometrie patří celá do jednoho poloprostoru.



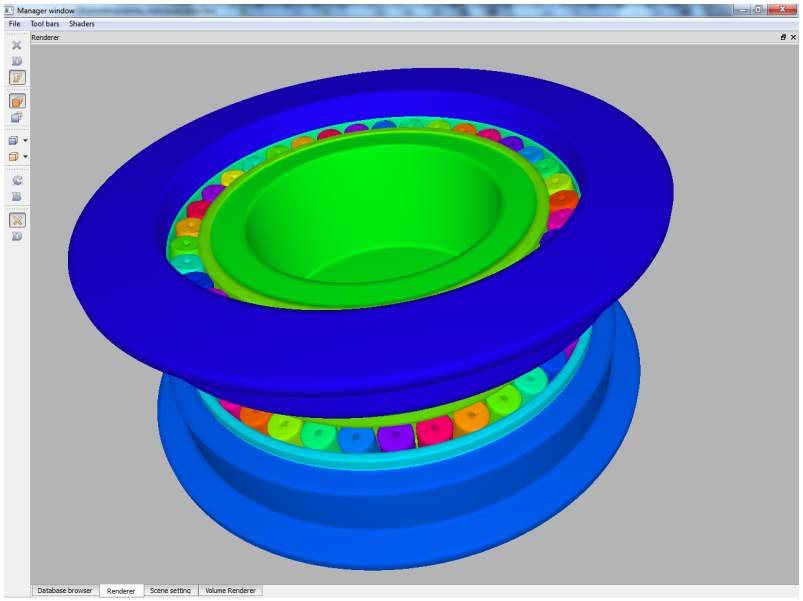
Obrázek 17: Graf výkonu řezu odstraňující pouze malou část modelu

Poslední test (Obrázek 18) je naopak varianta, při které je odříznuta velká část modelu. Zde se velmi projeví použítá datová struktura, díky které lze takto vynechat obrovské množství výpočtů. Výsledky i toto tvrzení potvrzují, když dochází k největším rozdílům mezi variantou s octree a bez ní.

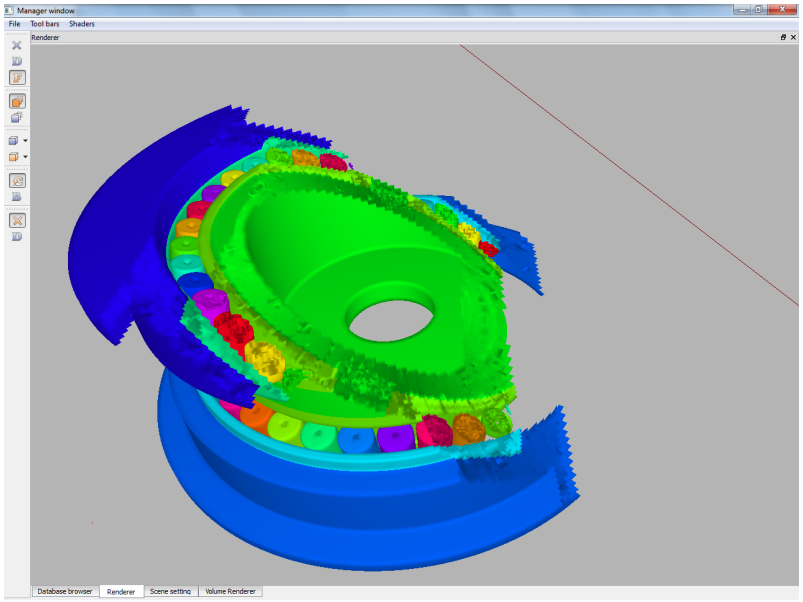


Obrázek 18: Graf výkonu řezu odstraňující velkou část modelu

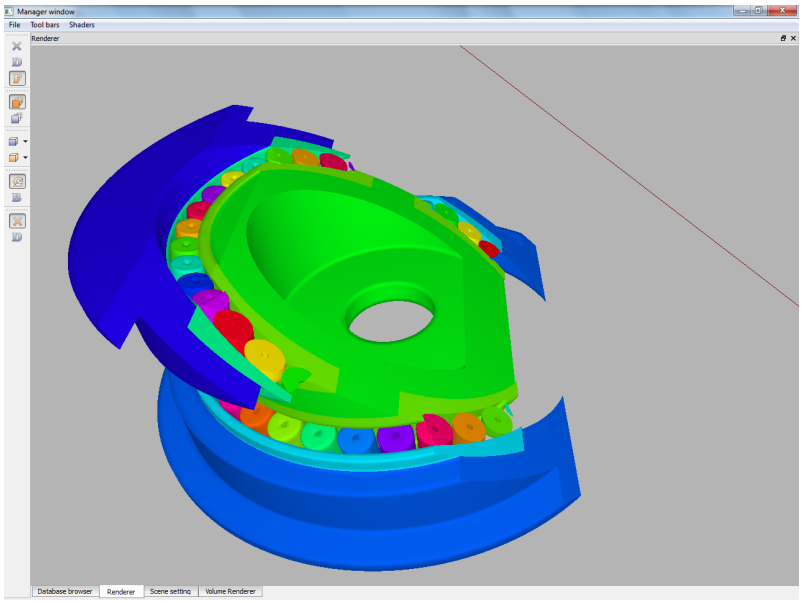
5 Ukázky aplikace



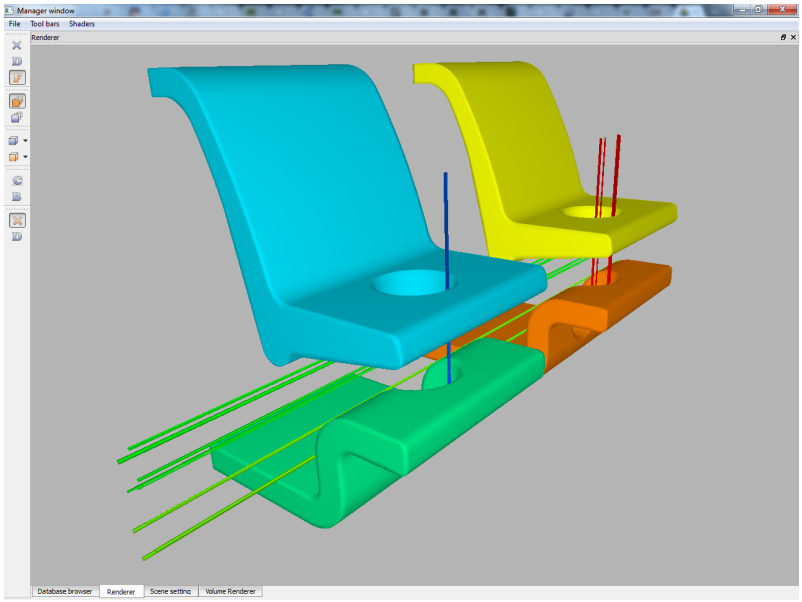
Obrázek 19: Model kuličkového ložiska s 659 820 elementy



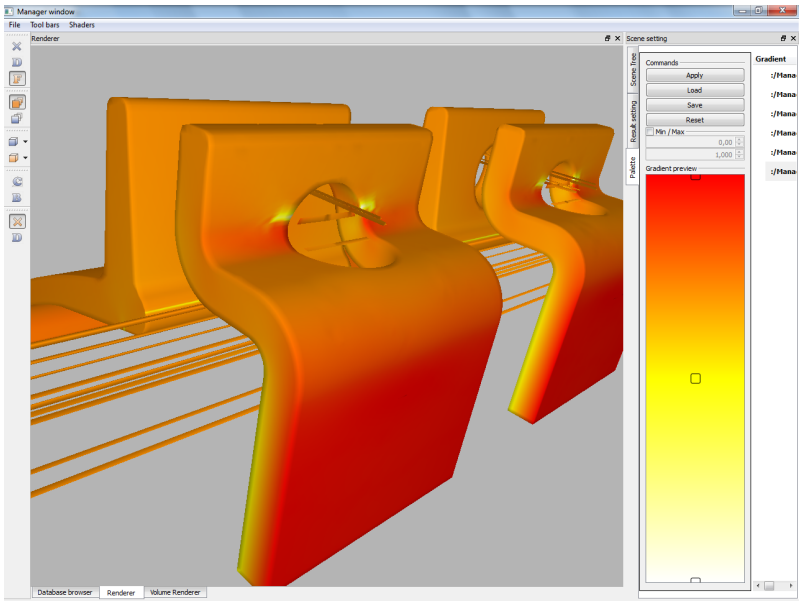
Obrázek 20: Řez modelem kuličkového ložiska s 659 820 elementy technikou *fast*



Obrázek 21: Řez modelem kuličkového ložiska s 659 820 elementy technikou *precise*



Obrázek 22: Model určitého držáku s 158 314 elementy



Obrázek 23: Model určitého držáku s 158 314 elementy a aplikovaným výsledkem konečných prvků

6 Splnění cílů

V práci byly probrány způsoby vizualizace metody konečných prvků, a to přes rasterizaci a volumetrické renderování. Zmíněno bylo i odkud a jak vznikají vstupní data této metody a co je jejich výsledkem. V implementační sekci se detailně vysvětlilo, jaké části aplikace jsem programoval já a jak jsem přitom postupoval. Na obrázku se znázornilo z čeho se každý vizualizovaný model skládá. Naznačeny byly poté některé zásadní problémy se správou těchto rozsáhlých dat a jaké kroky byly podniknuty k jejich eliminaci nebo alespoň zmírnění jejich vlivu. Konkrétně problémy s konečnou velikostí systémové paměti a rychlost alokace a dealokace paměti pro velké množství objektů, představovali závažné problémy, se kterými jsem si musel poradit. Nadefinovali se jednotlivá rozdělení modelu na pod-části a vysvětlila se jejich potřeba pro efektivní práci nad vizualizovaným objektem.

Zdůvodnilo se zvolení oktantového stromu jako prostorové datové struktury pro rychlejší výpočty řezů modelu. Bylo představeno, jak probíhá vkládání elementů do této struktury a jak se přes ní získávají řezy pro standartní i volumetrickou vizualizaci.

Jako poslední věc pro implementaci byla otázka použitých cizích knihoven pro usnadnění vývoje aplikace. Zejména výběr knihovny pro tvorbu uživatelského rozhraní a zvolení grafického API.

Nakonec se provedlo několik závěrečných výkonostních testů, které ukázali nárůst výkonu snížením potřebného času na výpočet řezů za různých podmínek. Ukázalo se, že za cenu preprocesingu, ve kterém se vytvoří octree, lze získat více jak 2x lepší časy výpočtů u odpovídajících si technik řezů.

Směr další práce by mohl směřovat k dalším optimalizacím pro vícejádrové procesory využitím některé z technik pro paralelizaci úloh. Rozdělení modelu na několik menších částí v podobě meshpartů tomu velmi napomáhá, jelikož se s nimi pracuje nezávisle na sobě. Nutné by bylo hlavně vyřešit více deserializací z disku současně.

7 Reference

[OpenGL] Oficiální stránky OpenGL, obsahují specifikaci standardu
`http://www.opengl.org/`

[Qt] Kompletní dokumentace ke knihovně Qt
`http://qt.nokia.com/`

[QGLViewer] Pomocná knihovna pro OpenGL okno v Qt
`http://www.libqglviewer.com/`

[Mac89] Richard H. MacNeal: *Finite Elements: Their design and performance*
Mechanical engineering, 1989

[Kau05] Arie Kaufman, Klaus Mueller: *Overwiew of volume rendering* (Academic Press 2005)
`http://www.cs.sunysb.edu/~mueller/papers/volvisOverview.pdf`

[Sam88] Hanan Samet: *Overwiew of quadtrees, octrees and related hierarchical data structures*
`http://www.cs.umd.edu/~hjs/pubs/Samettfcgc88.pdf`

[Gri87] Kurt R. Grice: *Finite octree meshing through topologically driven geometric operators*
`http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19880009737_1988009737.pdf`

[Tia03] Tiankai Tu, David R. O'Hallaron, Julio C. López: *The Etree Library: A system for manipulation large octrees on disk* (Červenec 2003)
`http://www.cs.cmu.edu/~euclid/etree-tr.pdf`

[Lan10] Frank C. Langbein: *Real-Time rendering: Polygon shading* (Leden 2010)
`http://www.langbein.org/publish/graphics/III/G-17-III_6-handout.pdf`

8 Obsah CD

/root	
abstract_cz.txt	abstrakt napsaný v českém jazyce
abstract_en.txt	abstrakt napsaný v anglickém jazyce
ReadMe.txt	obsah cd
/docs/	adresář dokumentů
dokumentace.pdf	elektronická podoba písemné části diplomové práce v pdf formátu
uzivatelska_prirucka.pdf	uživatelská příručka
/source/	složka se zdrojovým kódem mé práce